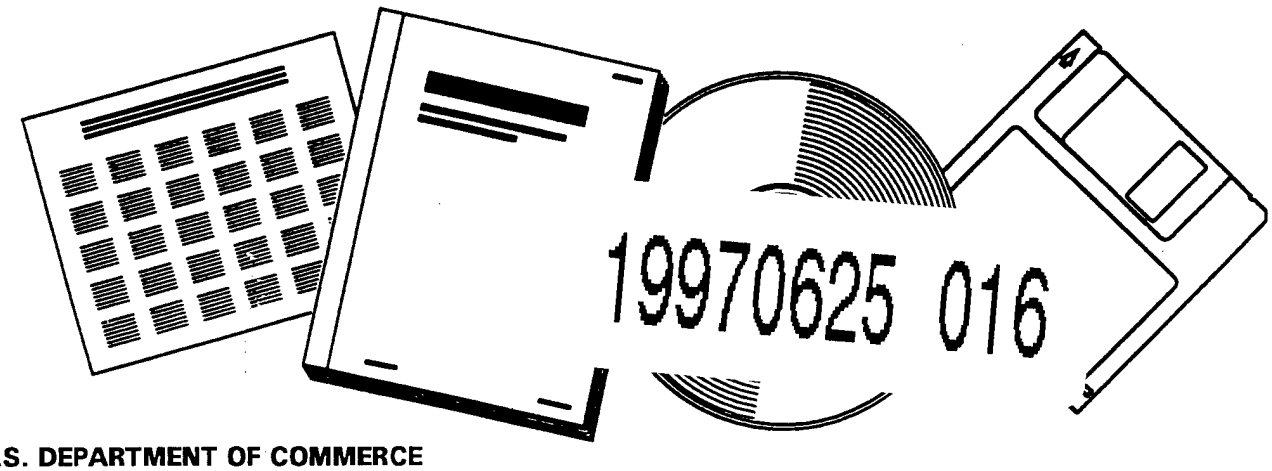
PB93-228724

THE FLUENT ABSTRACT MACHINE

THINKING MACHINES CORP.
CAMBRIDGE, MA

1987



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

The Fluent Abstract Machine

A.G. Ranade, S.N. Bhatt and S.L. Johnsson, Yale University

Thinking Machines Corporation
Technical Report Series

BA87-3

REPORT DOCUMENTATION PAGE

PB93-228724

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1987	Technical

4. TITLE AND SUBTITLE
The fluent abstract machine

5. FUNDING NUMBERS
ONR-N00014-86-K-0564
NSF MIP-8601885

6. AUTHOR(S)
A. Ranade, S. Bhatt, and S. L. Johnsson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Thinking Machines Corp.
245 First Street
Cambridge, MA 02142-1264

8. PERFORMING ORGANIZATION REPORT NUMBER
TMC-12

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ONR --Department of the NAvy
The Pentagon Washington, DC 20350
NSF -- 1800 G Street NW., Washington DC 21550

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The fluent abstract machine supports a very powerful programming model. In addition to arbitrary access patterns, the instruction repertoire of the fluent machine also includes the multiprefix operation and high-level set operations. The fluent machine consists of over one hundred thousand processors interconnected by a butterfly network. The efficiency of the fluent machine derives from a very simple router, which effectively eliminates the possibility of congestion. The routing hardware is extremely simple, inexpensive, and provably efficient.

14. SUBJECT TERMS
Basic Algorithms

15. NUMBER OF PAGES
22

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	SAR	SAR	SAR

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(if known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

The Fluent Abstract Machine

Abhiram G. Ranade
Sandeep N. Bhatt
S. Lennart Johnsson

Department of Computer Science
Yale University
New Haven CT 06520.

BA87-3

Underlying every general programming model is a shared address space. Every process can potentially access any object in this space in one step. While this allows tremendous expressive power, it poses an enormous challenge to the communications hardware. This conflict between ideal programming models and real architectures has traditionally been resolved by supporting a less general model which restricts the possible patterns of access.

The Fluent abstract machine supports a very powerful programming model. In addition to arbitrary access patterns, the instruction repertoire of the Fluent machine also includes the multiprefix operation and high-level set operations.

The Fluent machine consists of over one hundred thousand processors interconnected by a butterfly network. The efficiency of the Fluent machine derives from a very simple router, which effectively eliminates the possibility of congestion. The routing hardware is extremely simple, inexpensive, and provably efficient.

1 Introduction

We envisage building a Fluent machine with over one hundred thousand processors. Except for highly structured computations, such a large computer must necessarily spend a good deal of time communicating messages between its processors. As long as the total communication time does not swamp the total computation time, high performance is guaranteed.

Large parallel computers are also difficult to program. The situation becomes intolerable if the programmer must explicitly manage communication between processors. For this reason it is necessary to have a powerful programming model (abstract machine) which abstracts away concerns not directly relevant to the problem being solved. For overall performance, the abstract machine must be efficiently supported on the underlying machine.

Of the programming models proposed thus far, shared memory models have been the most attractive. The most general shared memory models in the literature, the concurrent-read concurrent-write parallel random-access machines (CRCW PRAMS)

allow an arbitrary number of processors to read or write a common memory location in one time step. Complex communications operations, broadcast and multicast for example, can be implemented in one step. Abstracting complex communications patterns into unit steps greatly simplifies the tasks of designing algorithms and writing programs. For this reason, CRCW PRAM models are favored over weaker abstract machine models for which most, if not all, of the programming effort is spent synchronizing the movement of data.

How do we implement a shared memory model on a machine with processors and memories distributed throughout an interconnection network? The solution is to devise an efficient router which emulates shared memory operations and hides details of the communications network from the user. This is precisely what recent machines such as the Thinking Machines Corporation's Connection Machine [8,9], the BBN Butterfly [2] and Monarch, the IBM RP3 [13], and the NYU Ultracomputer [6] aim to achieve.

These machines emulate abstract machines of varying generality and power. The Connection Machine CM2 has hardware support for concurrent read as well as concurrent write operations with combination. The Connection Machine and the NYU Ultracomputer/RP3 efficiently support the scan operation [4]. The Ultracomputer and RP3 also support the fetch-and-add operation, but the switching hardware is expensive and experiments reveal poor performance because of "hot spots" [11,14]. It thus becomes difficult to argue that the abstract machine operations are performed in unit time.

The Fluent abstract machine subsumes each of the abstract machines mentioned above. In fact, the *multiprefix* primitive of Fluent requires arbitrarily many primitive operations on the other abstract machines. The Fluent instruction set also includes basic set operations. With its rich instruction set, the Fluent abstract machine is readily suited as an intermediate language for compiling very high level languages.

The Fluent abstract machine can be supported efficiently and inexpensively in hardware. The heart of the Fluent machine is the router which is based on the recent work of Ranade [16]. In contrast with the Ultracomputer and RP3, the hardware requirements are minimal. More importantly, we can prove that each Fluent instruction is implemented quickly. This justifies our thesis that large Fluent machines will be less expensive, faster and easier to program than existing parallel machines.

The remainder of this extended abstract is organized as follows. Section 2 describes the Fluent abstract machine and contrasts it with other models. Section 3 outlines the implementation of the abstract machine on the butterfly network. Section 4 outlines a design for the routing switch. Section 5 describes the Fluent machine, presents results of timing simulations, and cost and performance estimates. Section 6 concludes with some of the important research issues that need further study, and outlines our ongoing work.

2 The Fluent Abstract Machine

This section describes the primitive instructions of the Fluent abstract machine, and contrasts the Fluent programming model with other models. In later sections we show how every instruction is supported efficiently in hardware. As a consequence, the time–complexity of a Fluent program can be easily estimated as the maximum number of primitive instructions executed by one processor.

The Fluent abstract machine has N (virtual) processors indexed $1, 2, \ldots, N$ which are connected to a shared memory holding M variables indexed $1, 2, \ldots, M$. The processors of the abstract machine operate synchronously in discrete time cycles. Every primitive instruction is executed in one time cycle; executing an instruction at time T (in the Tth time cycle) has the effect of changing the state that existed at the start of time cycle T.

The Fluent abstract machine is characterized by two types of primitives — *multiprefix* and *set operations*. The multiprefix operation is a fully general prefix operation and subsumes the fetch-and-op primitive on the NYU Ultracomputer [7], as well as the scan operation on the Connection machine [4]. Set operations are not supported as primitives on these machines. With its primitive set operations, the Fluent machine can be programmed at a very high-level of abstraction.

2.1 The Multiprefix Operation

The multiprefix operation has the form $MP(A,\ v,\ \otimes)$ where A is a shared variable, v is a value, and $\otimes$ is a binary associative operator. At any time step a processor can execute a multiprefix operation, with the constraint that if P_i and P_j execute $MP(A, v_i, \otimes_i)$ and $MP(A, v_i, \otimes_j)$, then $\otimes_i = \otimes_j$. The semantics of the multiprefix operator is as follows:

> At time T let $P_A = \{p_1 \ldots p_k\}$ be the set of processors refering to variable A, such that $p_1 < p_2 < .. < p_k$. Suppose that $p_i \in P_A$ executes instruction $MP(A, v_i, \otimes)$. Let a_0 be the value of A at the start of time T. Then, at the end of time cycle T, processor p_i will receive the value $a_0 \otimes v_1 \otimes \cdots \otimes v_{i-1}$ and the value of variable A will be $a_0 \otimes v_1 \otimes \cdots \otimes v_k$.

Thus, when a set of processors perform a multiprefix operation on a common variable, the result is the same as if a single prefix operation were performed with the processors ordered by their index. For example, suppose that processors numbered 25, 32 and 65 execute the instructions $MP(A, 4, +), MP(A, 7, +)$ and $MP(A, 11, +)$ respectively at time T, and suppose that variable A initially contains the value 5. Then, at the end of the Tth cycle, processor 25 will receive 5, processor 32 will receive 9, processor 65 will receive 16, and the variable A will equal 27.

The fetch-and-$\otimes$ operation [7] also calculates a set of prefixes, but the order of inputs is undetermined before execution. Multiprefix is a determinate implementation of the fetch-and-$\otimes$, and is more powerful. The *scan* operation [4] is a special

case of the multiprefix, one in which the set S includes all N processors. Scan does not allow multiple prefixes over all collections of disjoint subsets, whereas the multiprefix does.

For convenience we include two more primitives — READ and WRITE. READ(A) returns the value of A to the requesting processor. WRITE($A, v, \otimes$) is equivalent to $MP(A, v, \otimes)$ except that no value is returned to the processor executing the instruction. Both operations are special cases of multiprefix, as has been observed earlier [7].

2.2 A Fast Radix Sort using Multiprefix

In this section we present a radix sort based on the multiprefix instruction. The program is considerably simpler than Batcher's bitonic sort [1] and comparable in performance when the number of keys is very large.

When each key to be sorted is less than $\log N$ bits in size, fetch-and-add can be used to sort N keys in a constant number of steps. Unfortunately, this idea cannot be used iteratively to sort longer keys because the fetch-and-add, being non-deterministic, is not stable [4].

With the multiprefix we can implement a stable iterative radix sort. As we show below, N keys, each $k \log N$ bits long, can be sorted in $O(k)$ Fluent instructions. When k itself is small, the number of Fluent instructions executed is constant. In contrast, no other programming model supports such a concise sort even for short keys.

Theorem 1 *N keys, each of size $k \log N$, can be sorted in $O(k)$ steps on the Fluent abstract machine.*

Proof. We first describe a stable scheme for N keys of length $\log N$, one key per processor. The total number of distinct key values is N. Below we give the program for each processor. The keys to be sorted are stored in an array $KEY[*]$. The idea is to first count the number of occurrences[1] of $KEY(i)$ that lie in processors indexed less than i, then add to that the cumulative sum of the counts for keys less than $KEY[i]$.

```
SHORTSORT:
    COUNT[*]      := 0
    CUMULATIVE[*] := 0
    TEMP          := 0
    MP(COUNT[KEY[*]], 1, +)
    CUMULATIVE[*] := MP(TEMP, COUNT[*], +)
    return MP(CUMULATIVE[KEY[*]], 1, +)
```

[1]This simple histogram computation cannot be done in a constant number of steps on the scan-model [4].

Because the multiprefix operation is ordered by processor indices, the simple sort above is stable. We can iterate shortsort to sort larger keys in blocks. The primitive operation $LSBLOCK(w, j)$ below returns the least significant jth block of $\log N$ bits of location w, that is, bits $(j-1)\log N + 1$ through $j \log N$.

```
SORT:
    RANK[*]    := 0
    KEYPTR[*]  := *    (initialize pointer to self)
    FOR j=1 to k DO
        KEY[*]  := LSBLOCK(KEYPTR[*], j)
        RANK[*] := SHORTSORT[KEY[*]]
        KEYPTR[RANK[*]] := KEYPTR[*]
    ENDDO
```

2.3 Set Operations

Sets are a fundamental data abstraction. Traditionally, sets have not been supported as primitive objects, but instead have been built on top of lower level structures such as lists, arrays, trees and tables. The Fluent abstract machine includes set operations as primitives:

- INSERT (x, S) Insert element x into set S.
- DELETE (x, S) Delete element x from set S.
- MEMBER? (x, S) Is x an element of the set S?
- APPLY (S, f) Apply the function f to the elements of set S. Note that f may change the values of the elements in S.
- REDUCE (S, f) Evaluate f with arguments that are elements of S. Note that f must be a binary associative operator.

In addition, set union, intersection, difference, prefix, and enumerate are also supported.

Every Fluent processor can execute a set instruction, so that many sets can be manipulated simultaneously. For example, several processors may simultaneously insert elements, possibly into the same set. The result of concurrent set operations is as if the individual instructions were executed atomically in some arbitrary unspecified serial order. The implementation however is completely parallel, and provably efficient. The ability to simultaneously update multiple sets is costly on existing machines.

3 Implementing Fluent Instructions

This section describes how the Fluent abstract machine is implemented on the butterfly network. The routing algorithm used is extremely simple and provably efficient, and forms the basis of the Fluent machine proposed in Section 5.

3.1 The Fluent Network

The nodes of the Fluent machine are interconnected in the butterfly (FFT) pattern. There are 2^n nodes in each of $n+1$ levels, for a total of $N = (n+1)2^n$ nodes. Each node is labelled with a string $\langle c, r \rangle$ $(0 \leq c \leq n,\ 0 \leq r < 2^n)$ formed by concatenating the binary representations of the level number c and the index r of the node within the level. Each node $\langle c, r \rangle$ $(c < n)$ is connected by forward links to the nodes $\langle c+1, r \rangle$ and $\langle c+1, r \oplus 2^c \rangle$, where $\oplus$ denotes bitwise exclusive or. Each node (except for levels 0 and n) thus has four connections: two connections to the next higher level and two to the previous level.

Each node in the butterfly contains a processor, a memory module and 6 routing switches. Each switch has 2 inputs and 2 outputs. Every input into a switch enters a first-in first-out queue, which has the capacity to buffer a small number (2 or 3) of messages in transit.

3.2 The Address Map

The shared variables of a Fluent program are distributed among the local memories of the nodes using an appropriately chosen address map. If the Fluent program does not involve run-time address computation then the physical address of each shared variable can be embedded within the program of each processor. Otherwise, we must compute addresses quickly at run time.

We propose to distribute the M shared variables randomly among the processors, each processor being assigned M/N variables. With a random hash function, memory bottlenecks are unlikely because the accessed variables will be distributed throughout the network. Suppose that we have chosen such a hash function $\mathcal{H}$ [2]. This function maps a $\log M$ bit address to a $\log N$ bit node address. A second function $\mathcal{M}$ computes the address ($\log(M/N)$ bits) within the memory of node $\mathcal{H}(x)$. The physical address of shared variable x is given by the concatenation $\langle \mathcal{H}(x), \mathcal{M}(x) \rangle$.

[2] Our simulations show that simple first degree polynomials perform well in practice. A random $O(\log N)$ degree polynomial provably works well [10,16].

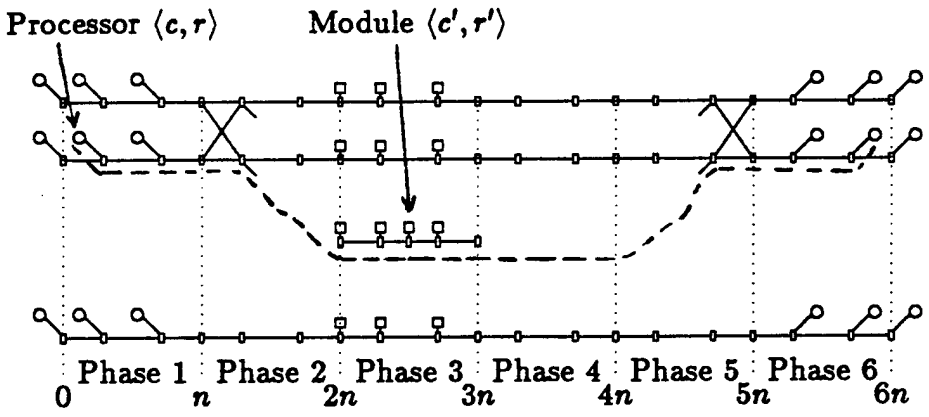


Figure 1: Logical Network

3.3 Message Structure and Path

Suppose that processor $\langle c, r \rangle$ wishes to access variable x. It generates a REQUEST, a message of the form $\langle \text{dest}, \text{type}, \text{data} \rangle$. The destination dest is $\langle \mathcal{H}(x), \mathcal{M}(x) \rangle$, the physical address of varible x. The type field denotes the kind of access requested, e.g. READ, WRITE, or MP. Other possible values include EOS or GHOST, which are used internally by the communication algorithm as we will see shortly. The REQUEST is injected into the network. It will reach node $\mathcal{H}(x)$ and return with the required data.

The path from node $\langle c, r \rangle$ to node $\mathcal{H}(x) = \langle c', r' \rangle$ and back involves 6 phases through the butterfly. Every other phase is a forward phase, and these are interleaved with backward phases. Figure 1 shows the 6 phases.

In the first phase, the message issued at node $\langle c, r \rangle$ is directed to node $\langle n, r \rangle$. In Phase 2, the message follows the unique (backward) path in the butterfly from node $\langle n, r \rangle$ to node $\langle 0, r' \rangle$. This path is determined at each switch by looking at the appropriate bit of dest. In Phase 3, the message reaches the node $\langle c', r' \rangle$, where it acquires the required data. The next 3 phases simply retrace the path traced thus far, back to the source processor $\langle c, r \rangle$. The access is now complete.

For convenience, we describe the routing mechanism in terms of the logical network of Figure 1 instead of the butterfly. The correspondence between the two is clear and each butterfly node does the work of 6 switches in the logical network.

3.4 How to Combine Messages

At the heart of the Fluent machine lies the routing strategy [16]. The key idea is a simple way of combining instructions that reference a common variable. Consider the case when several processors READ a common variable. The paths of these messages form a tree, as in Figure 2. Each message moves along the directed path

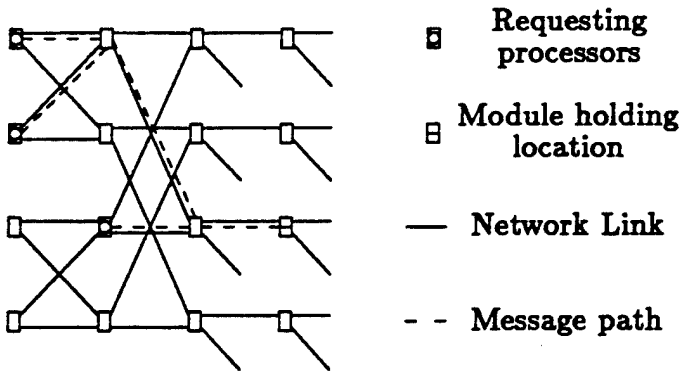


Figure 2: Message paths to a common location form a tree

from its source to the destination.

There is, however, no need to send more than one request along any branch of this tree. Each tree node forwards a request only when it "knows" that no future incoming request will have the same destination. The key idea here is that each node forwards requests in ascending order of destination addresses. Each node receives messages along two incoming edges and places them into the corresponding FIFO queues. At each step the node compares the destination addresses of the messages at the heads of the two queues. The message with the smaller destination address is transmitted forward. If both messages are destined for the same location, they are combined and only one request is sent out. Finally, if only one queue has a message waiting and the other queue is empty, no message is sent out. (If the message were sent, the next message along the other edge could conceivably have a smaller destination, thus violating the sorting requirement).

In our snapshot at time T, node A in Figure 3 selects the message destined for location 35. Then it waits until the message to location 48 arrives, at which point it discovers that the messages at the heads of both the queues are to location 48, and can be combined.

3.4.1 Reply routing

How do we return the data to all requesting processors? The reply message, upon reading the data, returns backwards along each edge of the tree and reaches every requesting processor. For the backrouting we only need to store two *direction bits* at each node. The bits say whether the request came along the top branch, the bottom one, or along both. Since messages are kept sorted throughout the six phases, *replies at each node arrive in the same order as the requsts were sent out.* Therefore, the direction bits can be stored in a 2-bit wide FIFO queue. This simple idea is more efficient than the associative memories proposed earlier [7].

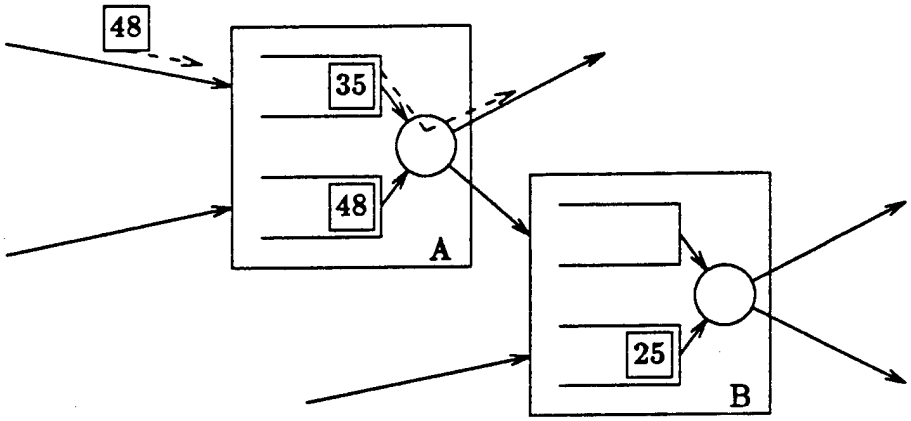


Figure 3: Combining Messages by Merging Streams

3.4.2 Ghost messages

The simple idea of keeping message streams sorted has one deficiency. Consider Figure 3 again. At time T, processor B cannot transmit the message it holds for location 25, because it does not know what will arrive on the top link. However, when A selects the message to location 35 for transmission, it can send a *ghost* message labelled 35 to B. When B receives the ghost message, it knows that future messages along that edge will be destined for locations greater than 35. Therefore, at the next time step B can forward the message waiting in the lower queue.

Ghost messages notify nodes of the minimum location to which subsequent messages can be destined. Ghosts are not used for any other purpose, they "keep the system fluent."

3.4.3 Flow control

It is possible that a switch S is ready to transmit a message forward but the input queue for next switch is full. When this happens, S simply retains the message and tries in the next clock cycle. Of course, if the message S tried to transmit was a *ghost*, it can be dropped without any loss of information.

Many routing algorithms which adopt such a holding policy give poor performance because congested buffers back up buffers upstream. For our algorithm the probability of such degradation is provably miniscule, and the algorithm is always deadlock-free.

3.4.4 Termination

Immediately following a request, each processor also issues an end-of-stream EOS message. The dest field of every end-of-stream message is ∞. An EOS notifies a switch that no more requests will follow. The switch can now safely forward the requests on

the other edge, and eventually forward the EOS messages themselves. EOS messages form a wavefront which guarantees that every instruction will terminate.

3.4.5 Performance

Following Ranade [16], we can show that this routing algorithm is close to optimal.

Theorem 2 *Assuming a perfect random address map, the probability that any memory reference takes more than* $15 \log N$ *steps is less than* N^{-20}.

Every routing algorithm must take at least $4 \log N$ steps. Observe that the provable performance is only slightly far from this lower bound, and considerably faster than previous algorithms for routing on butterflies of reasonable size.

Figure 7 gives timing results from simulations of the routing algorithm. We experimented with a number of different memory access patterns, e.g. matrix access, trees of different types, shuffles, random permutations etc. In no case was the time taken more than $11 \log N$, even with queues of size 2. Increasing queue size did not appreciablly affect performance. We found that simple hash functions (shared variable x mapped to physical address $ax + b \bmod M$) were satisfactory. Section 5.1 describes more simulation experiments.

3.5 Multiprefix instructions

We first describe the implementation for fetch-and-add proposed in [7]. Let s be an arbitrary switch in phase 1 (or 2). Suppose that the messages at the heads of the queues are $m_1 = \langle l, \text{fetch-add}, v_1 \rangle$ and $m_2 = \langle l, \text{fetch-add}, v_2 \rangle$ respectively. As shown in [7] the switch must forward a message $m = \langle l, \text{fetch-add}, v_1 + v_2 \rangle$ in place of m_1 and m_2. If the reply to m is a value v, then the corresponding switch in phase 6 (or 5) returns v as a reply to m_1, and $v + v_1$ as a reply to m_2. Thus the switch must remember the value v_1 received on its top queue for each pair of fetch-and-add messages that it combines.

Notice that this is equivalent to a serial execution of the message received on the top input (m_1) before the message received on the bottom input (m_2). Thus if we ensure that messages received on the top input always originate in a processor with a smaller number than those received at the bottom input, we effectively have an implementation for the multiprefix operation, with addition replaced by the prefix operator. We show how to do this by numbering the processors appropriately.

Theorem 3 *The multiprefix operation will, with overwhelming probability, terminate in* $O(\log N)$ *steps.*

Proof: We present the required numbering for the processors and switch inputs. Processor $\langle c, r \rangle$ is numbered $nr + c$. A switch $\langle c, r \rangle$ in phases 1 or 2 receives its

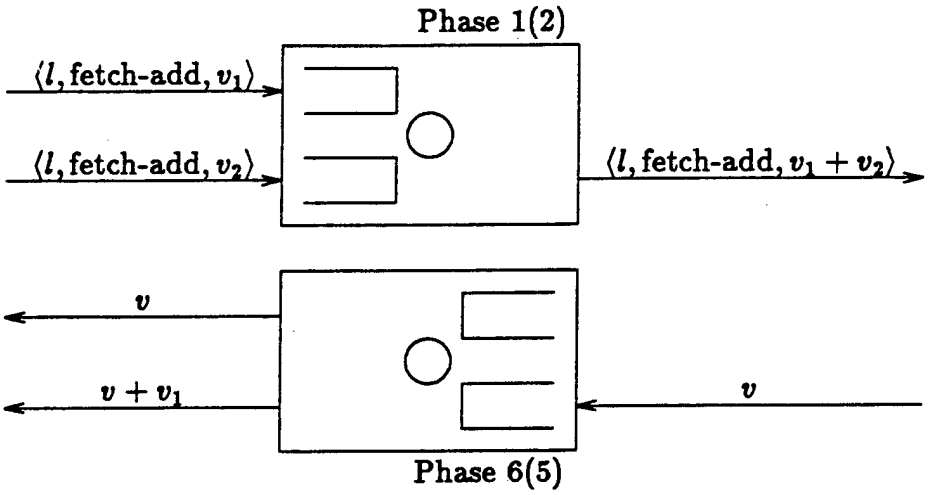


Figure 4: Fetch-and-add

inputs i_0, i_1 from switches $\langle c-1, r_0\rangle$ and $\langle c-1, r_1\rangle$ respectively. If $r_0 < r_1$, then we shall label i_0 as the top else we label i_1 as the top. ∎

As noted earlier, the only extra requirement over a read instruction is that, in addition to the two direction bits, each switch must remember a value (*partial sum*) for every combination that occurs at that switch. Figure 5 shows a pair of switches with the required queues.

3.6 Processor synchronization

"It is always 4 o' clock here," said the March Hare to Alice.
—*Lewis Carroll,* Alice in Wonderland

We use EOS messages to implement a distributed global clock. Recall that one EOS message per instruction passes through each switch. By maintaining a count of the number of EOS messages that have passed through, each switch keeps its version of the global time.

Different switches may indeed have different counts or versions of the global time, but that is perfectly alright. If two instructions access a common location in the same time step, then the one that arrives first will have to wait for the slower one to reach an intermediate switch for combination. Because we keep messages sorted by tag, and we guarantee that only one request for access will be passed into the memory module which holds the variable, the effect is the same as if all the processors were operating synchronously. For example, our implementation guarantees that for the code of figure 6 processor 1 and 2 will respectively read 10 and 20, provided no other processor writes a and b in the meantime. This is guaranteed in spite of the fact that both processors might issue all 3 instructions without waiting for any to complete. This is a very strong synchronization condition requiring special primitives on most other programming models.

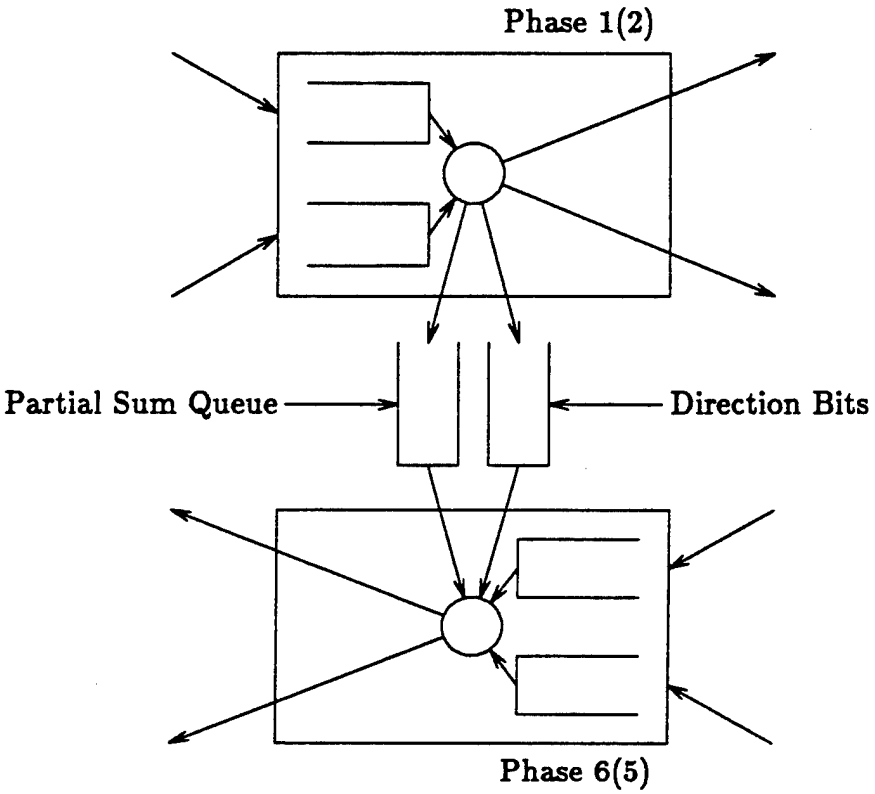


Figure 5: Internals of a pair of switches

TIME	PROCESSOR 1	PROCESSOR 2
1	A=20	B=10
2	Read B	Read A
3	A=30	B=40

Figure 6: Synchronization Guarantee

This implementation also allows each processor to stop the global clock if necessary, if it detects an error for example. This is done by withholding the end-of-stream message.

For lack of space we will not describe how set operations are implemented. The interested reader is referred to [15].

4 The Routing Switch

In this section we outline a bit-serial design for the routing switch and estimate its layout requirements. The design extends to wider data paths in a straightforward manner.

Although Section 3 presented the routing algorithm with the implicit assumption that messages were transmitted in atomic packets, this is not necessary. In particular, each message can be transmitted bit-serially in a pipelined manner. This is analogous to the wormhole router of Dally and Seitz [5]. Message transmission can be pipelined because:

1. Address comparison can be done bit-serially, provided the addresses are received most significant bit first.
2. Message combination can be done bit-serially; for operators like +, the data must be transmitted least significant bit first. Also see on-line arithmetic [17].
3. When a message leaves a switch, the corresponding GHOST message (whose dest is identical to the real message) can be generated bit-serially.

Each message is transmitted with the dest field first (most significant bit leading), followed by the type field, and finally the data field (least siginficant bit leading). A switch begins operating when: (1) each input queue contains at least one message, and (2) the input queues of the receiving switches are not full.

We now describe the operation of a switch in phase 2. Switches in other phases can be specified similarly.

1. **Transmit dest:** The minimum of the destinations of the two messages in the input queues is transmitted along both outputs. The minimum is discovered only after the transmission, so till then both destinations must be retained in the input queues.
2. **Transmit type:** While transmitting the destination, the switch detects which output link the request must be routed on. This requires checking one fixed bit in the destfield. The type of the message with the minimum destination is transmitted on the that output, while on the other, type GHOST is transmitted.

3. **Transmit data:** This is relevant for messages like MP-⊗ or WRITE. In either case, the message type indicates how messages must be combined when necessary. Again, the data fields can be combined and transmitted as they arrive.

The ability to pipeline messages speeds up message delivery considerably when there are no queueing delays. The message delivery time reduces from (network latency) × (message length) to (network latency) + (message length). We expect the latency of each switch to be about 4 (message enters an input queue, passes through the ALU, is sent to the output queue, and then transmitted), giving a total latency of $4 \times 6n$ for the logical network. Assuming 100 bit long messages and 4-bit wide data paths, the time for a 13 dimensional butterfly is $(4 \times 6 \times 13) + 100/4 = 337$ steps.

We now estimate the area requirements for the routing switches per node. Each switch consists of message queues, an ALU (for address comparison, message combination, etc.), counters to maintain the message FIFO queues, memory for storing partial sums, and direction bits for reply routing. In the following we assume that messages are 100 bits wide, and that partial sums are 64 bits wide.

Switches in phases 2 and 5 have two input queues, while others only have one input queue. The total number of message queues per node is therefore 8. Simulations (section 5.1) indicate that for 100,000 node machine each message queue need hold only 3 messages. The total memory requirement for message queues thus equals $8 \times 3 \times 100 = 2400$ bits, or roughly 1.2 $M\lambda^2$(at $500\lambda^2$per bit[3]).

Simulations also strongly indicate that no switch will ever transmit more than 40 messages along its outputs. For reply routing we need 2 bit wide direction queues, and 64 bit wide partial sums. Long partial sum queues are maintained only in phase 2 so that the total memory requirement adds up to $40 \times 64 + 6 \times 2 \times 40 = 3040$ bits, or 1.52 $M\lambda^2$.

Each queue requires 3 counters, except for the message queues which require 4. Assuming 8 bit wide counters, the total memory is 424 bits. With 3000 λ^2per counter bit, total area requirement is 1.28 $M\lambda^2$.

Assuming 8 bit wide data paths, each ALU requires around 1.2 $M\lambda^2$, for a total of 7.2 $M\lambda^2$per node.

The total area requirement is thus approximately 11.2 $M\lambda^2$. Including miscellaneous overhead, 15 $M\lambda^2$is a conservative estimate for 6 switches per node.

5 The Fluent Machine

This section presents an outline for a Fluent machine which can be constructed within the next few years with conservative technology. Table 1 summarizes our

[3]The estimates for the different components are from [12].

Feature size for VLSI	1μ ($\lambda = 0.5\mu$)
Chip size	$100mm^2 = 400\ M\lambda^2$
Pins per chip	≈ 150
Printed circuit boardsize	.5 m × .5 m
Off board connections	512

Table 1: Technology for the Fluent-I

switches	30 $M\lambda^2$
2 32-bit RISC Processors	40 $M\lambda^2$
Floating point unit	100 $M\lambda^2$
128 Kbytes memory per processor	200 $M\lambda^2$
Total area requirement per chip	370 $M\lambda^2$

Table 2: Chip Specification

assumptions about the technology available. Needless to say, breakthroughs in packaging technologies will have the largest impact.

The Fluent-I is organized as a 13–dimensional butterfly, with 2^{13} nodes in each of 14 ranks for a total of 114,688 nodes. These nodes are divided into 256 boards, each housing a 6–dimensional butterfly. The network is partitioned into 2 planes of boards, arranged in the manner suggested by Wise [18]. Each board has 448 nodes, divided among 224 chips, with 2 nodes per chip. In addition to the 2 processors, each chip also has routing switches for the two nodes, one floating point unit (multiplier and adder), and memory. Table 2 summarizes the breakup of chip area, using estimates as in the previous section.

Data paths between nodes vary in width depending on whether the path is on-board or across boards. Each board has 128 4-bit wide data paths out (64 nodes in the last rank of a 6-dimensional butterfly, each with 2 forward links). On-board paths are 8 bits wide. The butterfly can be partitioned so that each chip requires 16 data paths so that 128 pin connections suffice.

This variation in data path widths was not considered in the previous sections. The performance of the routing algorithm changes somewhat with narrow channels. The off-board channels also have to be multiplexed over the 6 phases of the logical network, while on-board channels are replicated. At worst one would estimate that the 4-bit wide off-board channels would slow the system by a factor of 12 (the other channels are 8 bits wide), but our simulations show that this is wildly pessimistic.

Processors	114,688
Floating Point Units	57,344
Memory	16 Gbytes
Cycle time	50 ns
Peak Floating Point Rate	2.3 Tflops

Table 3: Fluent-I Highlights

5.1 Simulation Results

We performed timing simulations of the communication network on the Connection Machine. The objectives were to observe the sensitivity of the routing scheme to variations in queue size, address maps, and memory reference patterns. A final objective was to study the effect of using multiplexed, narrower offboard channels.

Our conclusions in brief:

1. Simple hash functions perform well. We tried various linear congruential maps: variable x placed in location $ax + b \bmod M$, where M, the size of the address space, is a prime, and a and b are constants.
2. Routing time varies little with access pattern. We tried several patterns: matrix access, binary trees, shuffle permutations, random accesses etc. Random patterns took slightly longer in all cases.
3. Concurrent access is faster than exclusive access. The extreme case is when all processors read the same variable. The number of steps reduces from 154 (see Figure 7) to 85 because there is no buffering delay. This assumes that messages are no wider than channels (see below for further discussion).
4. Queue–size 3 is adequate. While queue size 1 degrades performance drastically, queue sizes 2 or more give similar performance.
5. Figure 7 plots the routing time when off-board channels are multiplexed, with *narrowness* being the ratio of the width of the offboard channels to the on-board channels. Switches in lower phases are given higher priority in accessing channels. Each channel first allows phase 0 messages to pass, followed by phase 1 messages, and so on. From the plot we can conclude that the performance degrades by a factor of 1.7 over the ideal case (no narrow channels and no multiplexing). The time goes up from 154 steps as in Figure 7 to about 260 steps (extrapolated for 114,688 processors from Figure 7).

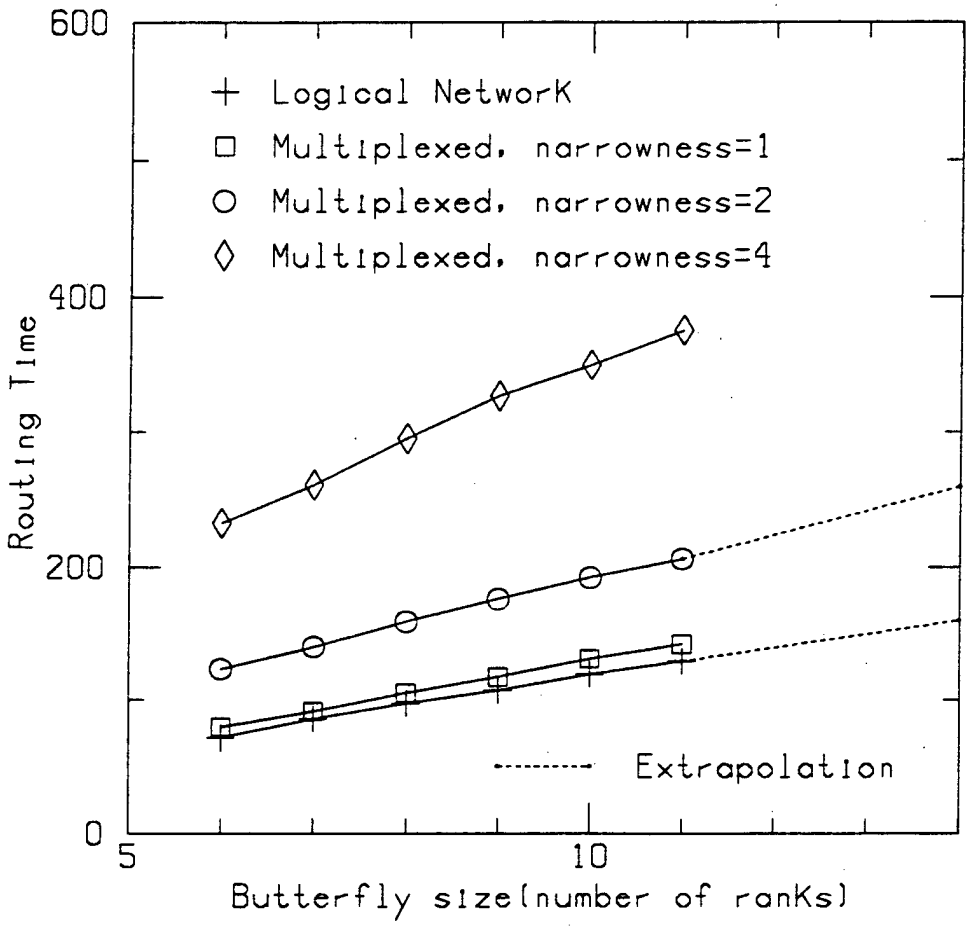


Figure 7. Average routing time
(50 randomly chosen access patterns)

5.2 Router performance

Suppose that messages are 100 bits long (64 data bits, 32 address bits, and 4 type bits). If every channel was 8 bits wide, sending a message across one link would require $100/8 \approx 13$ steps. From the results of the previous section we can therefore estimate that, with narrow channels and multiplexing, an arbitrary permutation can be routed in $260 \times 13 = 3380$ steps. With a 50 nanosecond clock rate, the time is about 169 μsec.

If all processors access a single variable, then the time is just 337 cycles (section 4), or about 17 μsec.

As an example, suppose that we wish to sort 16-bit numbers, with 32K numbers in each node. There are roughly 3.5 billion numbers being sorted. On the Fluent machine, we only need one iteration of the procedure SHORTSORT from Section 2. For each number being sorted, 3 shared memory instructions are executed (the others are local). However, the instructions can be packed into 50 bit messages. The total number of steps required is $3 \times 32K \times 3380/2$, or about 169 million for a total time of 8.5 seconds. If the numbers are 32 bits long, the time is about 17 seconds. Note that this is the time to sort the entire contents of memory.

5.3 Structured Computations

Much work has been done on mapping structured computations onto butterfly networks. These computations do not need the generality of shared memory. Better performance can be achieved by direct nearest neighbor communication rather than routing. This allows us to utilize the floating point capabilities of the machine more efficiently.

Table 3 presents performance estimates for two structured problems: FFT and Matrix multiplication. We considered a 2^{30} point complex FFT, and used the standard mapping. We obtain a performance of between 1.2 Tflops and 2 Tflops depending upon the assumptions made about local memory bandwidth. Batcher's bitonic sort (N numbers) on the butterfly takes $2\log^2 N$ steps. With $32K$ 16 bit numbers per node, and each communication step requiring 4 cycles, the total time is $4 \times 2 \times 289 \times 32K \approx 75M$ cycles. At 50 ns clock this gives a time of 3.7 seconds. While this estimate is lower than that of the shared-memory radix sort, extracting the extra peformance requires non-trivial and tedious low level fine tuning.

Besides nearest neighbor communication, performance gains can also be achieved by partitioning structured problems into blocks, and doing block computations locally within each node. This reduces the number of shared memory instructions. For matrix multiplication there are no good mappings into the butterfly [3]. Instead, we partition a large matrix into block submatrices, each of which is stored in one node. Instead of mapping blocks randomly to nodes, we use a simple hierarchical approach: decompose the matrix into large blocks, and map these into random boards. Next, decompose the large blocks into smaller blocks and map

Estimated multiprefix time	169μsec
Radix Sort $3.5 \cdot 10^9$ 16 bit numbers	8.5sec
Bitonic Sort $3.5 \cdot 10^9$ 16 bit numbers	3.7sec
Matrix multiplication	0.8 Tflops
FFT	1.2 Tflops

Table 4: Fluent-I Performance

them randomly into nodes. This allows us to exploit locality at the processor and board levels, and reduces the communication load on the off-board channels.

6 Conclusions and Extensions

Powerful models of parallel computation need neither be expensive nor slow – this is what we wish to demonstrate by building a Fluent parallel computer. In this extended abstract we have presented the Fluent abstract machine which is more powerful than any other abstract shared memory model, and shown that it can be implemented inexpensively on the Fluent machine.

We are continuing simulation experiments. By programming different applications we hope to get more insight into the expressive power of the Fluent programming model. We also expect to identify various tradeoffs, and adjust design parameters accordingly. For example, by providing even wider data paths on board, at the expense of reducing the number of switches per node (by multiplexing them) we expect that overall performance can be improved.

In this abstract we have not considered many issues in processor/chip design, and have mostly presented very conservative estimates for area requirements. We expect to begin detailed design of the router and communications harware following our experiences with the simulator. Future work will throw more light on issues such as SIMD vs. MIMD organization, processor complexity/wordlength, and operating system issues.

Acknowledgements

We thank Nick Carriero, David Greenberg, Tom Leighton, Charles Leiserson, and the students of the Fall '87 CS445 (Parallel Algorithms and Architectures) class at Yale for stimulating discussions. We are especially grateful to Carlton Geckler for helping us access the Connection Machine at NPAC, Syracuse University. Lennart Johnsson and Abhiram Ranade were supported by ONR grant N00014-86-K-0564. Sandeep Bhatt was supported by NSF grant MIP 8601885.

References

[1] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Comp. Conf.*, pages 307–314, 1968.

[2] *Butterfly Parallel Processor Overview.* BBN Laboratories Inc., 1985.

[3] S.N. Bhatt, F.R.K. Chung, J-W. Hong, F.T. Leighton, and A.L. Rosenberg. Optimal simulations by butterfly networks. In *Proceedings of the ACM Symposium on Theory of Computing*, 1988. to appear.

[4] Guy Blelloch. Scans as primitive parallel operations. In *Proceedings of the International Conference on Parallel Processing*, pages 355–362, 1987.

[5] William Dally and Charles Seitz. *Deadlock Free Message Routing in Multiprocessor Interconnection Networks.* Technical Report 5206:TR:86, California Institute of Technology, 1986.

[6] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32:175–189, February 1983.

[7] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Coordinating large numbers of processors. In *1981 International conference on Parallel Processing*, 1981.

[8] W. Daniel Hillis. *The Connection Machine.* The MIT Press, 1985.

[9] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *CACM*, 29(12):1170–1183, December 1986.

[10] Anna Karlin and Eli Upfal. Parallel hashing - an efficient implementation of shared memory. In *Proceedings of the Symposium on Theory of Computing*, 1986.

[11] Gyungho Lee, Clyde P. Kruskal, and David J. Kuck. The effectiveness of combining in shared memory parallel computers in the presence of 'hot spots'. In *Proceedings of the International Conference on Parallel Processing*, pages 35–41, 1986.

[12] John Newkirk and Robert Mathews. *The VLSI Designer's Library.* Addison-Wesley Publishing Co., 1983.

[13] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of International Conference on Parallel Processing*, pages 764–771, 1985.

[14] G. F. Pfister and V. A. Norton. Hot-spot contention and combining in multistage interconnection networks. In *Proceedings of International Conference on Parallel Processing*, pages 790–797, 1985.

[15] Abhiram G. Ranade. Fluent Data Structures for Sets. in preparation.

[16] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1987. Also available as Yale Univ. Comp. Sc. TR-578.

[17] K. S. Trivedi and M. D. Ercegovac. On-line algorithm for division and multiplication. *IEEE Transactions on Computers*, C-26:681–687, July 1977.

[18] David S. Wise. Compact layouts of Banyan/FFT networks. In H. T. Kung, Bob Sproull, and Guy Steele, editors, *Proceedings of CMU conference on VLSI systems and computations*, pages 186–195, 1981.

Reproduced by NTIS

National Technical Information Service
Springfield, VA 22161

This report was printed specifically for your order from nearly 3 million titles available in our collection.

For economy and efficiency, NTIS does not maintain stock of its vast collection of technical reports. Rather, most documents are printed for each order. Documents that are not in electronic format are reproduced from master archival copies and are the best possible reproductions available. If you have any questions concerning this document or any order you have placed with NTIS, please call our Customer Service Department at (703) 487-4660.

About NTIS

NTIS collects scientific, technical, engineering, and business related information — then organizes, maintains, and disseminates that information in a variety of formats — from microfiche to online services. The NTIS collection of nearly 3 million titles includes reports describing research conducted or sponsored by federal agencies and their contractors; statistical and business information; U.S. military publications; audiovisual products; computer software and electronic databases developed by federal agencies; training tools; and technical reports prepared by research organizations worldwide. Approximately 100,000 *new* titles are added and indexed into the NTIS collection annually.

For more information about NTIS products and services, call NTIS at (703) 487-4650 and request the free *NTIS Catalog of Products and Services*, PR-827LPG, or visit the NTIS Web site **http://www.ntis.gov**.

NTIS
Your indispensable resource for government-sponsored information—U.S. and worldwide